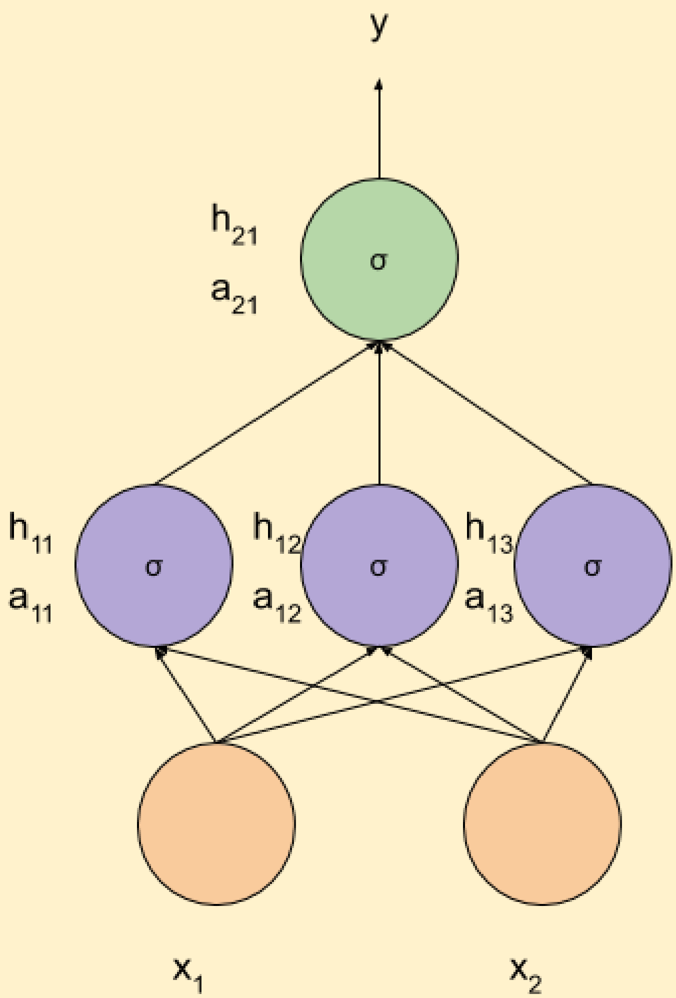
PadhAI: Activation Functions & Initialization Methods

One Fourth Labs

Xavier and He Initialization

Why shouldn't you initialise all the weights to large values?

1. Consider the following neural network that uses the logistic activation function



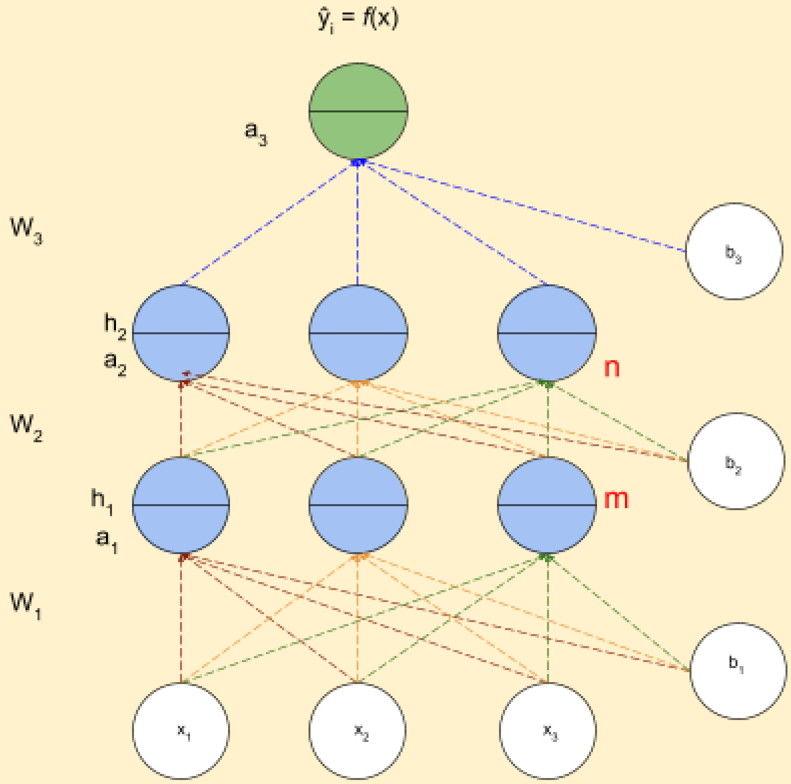
   a. $a_{11} = w_{11}x_1 + w_{12}x_2$
   b. $a_{12} = w_{21}x_1 + w_{22}x_2$
   c. Here, input values are normalised (0-1) and the weights are initialised to large values
   d. This would result in the function attaining saturation.
   e. Thus, a few noteworthy points are:
      i. Always normalise the inputs (should lie between 0 to 1). If not, they too could contribute to saturation if their values are very large or small
      ii. Never initialise weights to large values
   f. To reinforce the points we made earlier about initialisation
      i. Never initialise all weights to 0
      ii. Never initialise all weights to the same value
      iii. Never initialise all weights to large values

PadhAI: Activation Functions & Initialization Methods

One Fourth Labs

---

2. Let's look at a sample neural network and consider some alternate initialisation methods



3. **Xavier initialisation**:
   a. Consider $a_{21} = w_{21}h_{11} + w_{22}h_{12} + ... + w_{2m}h_{1m}$
   b. Here, as the number of input neurons increase (m>>0), the value of a_{21} could potentially be very high. Thus to avoid saturation due to a extremely large value, it is recommended that the weights scale inversely with the number of input neurons, i.e. $w \propto \frac{1}{m}$
   c. The python implementation is as follows

```
W_2 = np.random.randn(num_in, num_out) / sqrt(num_in)
    # This term is an m x n matrix     divided by sqrt(m)
```

   d. Here, np.rand.randn gives a value between 0-1 from the normal distribution. This m x n matrix is then divided by the sq.root of m. This prevents a_{21} from blowing up to a very large value. Used in the case of **tanh and logistic activations**
4. **He Initialisation**:
   a. It is used for ReLU and Leaky ReLU
   b. Here is the python implementation

```
W_2 = np.random.randn(num_in, num_out) / sqrt(num_in/2)
    # This term is an m x n matrix     divided by sqrt(m/2)
```

   c. Here, we divide by $\sqrt{m/2}$ because of the rough intuition that in ReLU, around half the neurons die during training.